Making an Interpreter

Because why not?

A Talk by

A Sachin
Soorya Narayan

Intro

- Why create a language?
- Program Translation: Compilers vs Interpreters

What we intend to do today

- Roadmap of the translation process
- Our language specification!

Lexing

- Crash course on regular expressions

Parsing

- Grammars and Abstract Syntax Trees

Introduction

1. Why create a language?

Introduction

Why create a language?

To demystify the magic under the hood

- Remove the magic from the compilation process
- Compiler development covers a wide range of topics from hardware to high level math
- A great opportunity to become familiar with concepts such as recursion, memory management, and data structures among other topics

Introduction

Why create a language?

To implement new design concepts

Dynamic vs Static typing

Declarative languages

1. Functional programming languages
2. Logic programming languages
3. Constraint based computing

Dynamic/Scripting languages

Event Driven and many more!

Introduction

Why create a language?

Little languages (aka Domain specific languages) are everywhere!

You would come across thousands of little languages for every mainstream language out there. These are tailored for very specific tasks

i. Markup languages like markdown
ii. Shell languages like bash
iii. Domain-specific language like Solidity for Ethereum

Introduction

Why create a language?

The real reasons :)

Bragging rights!

Proven trick to combat boredom!

Introduction

1. Why create a language?
2. Program translation: Compilers vs Interpreters

Introduction

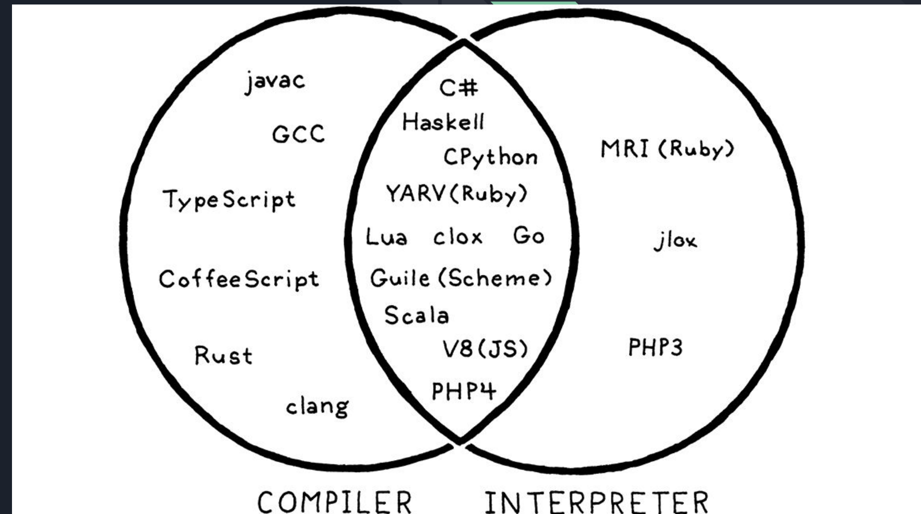
Program translation: Compilers vs Interpreters

Compilers

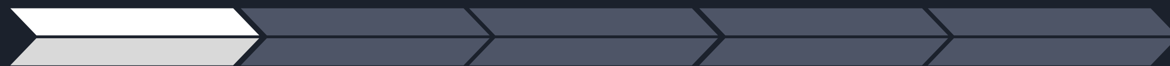
When we say a language implementation "is a compiler", we mean it translates **all of the source code all at once** to some other form but **doesn't execute it**.
The user has to take the resulting output and run it themselves.

Interpreters

Conversely, when we say an implementation "is an interpreter", we mean it takes in source code and **executes it line by line immediately without returning the intermediate representation** to the user.
It runs programs "from source".



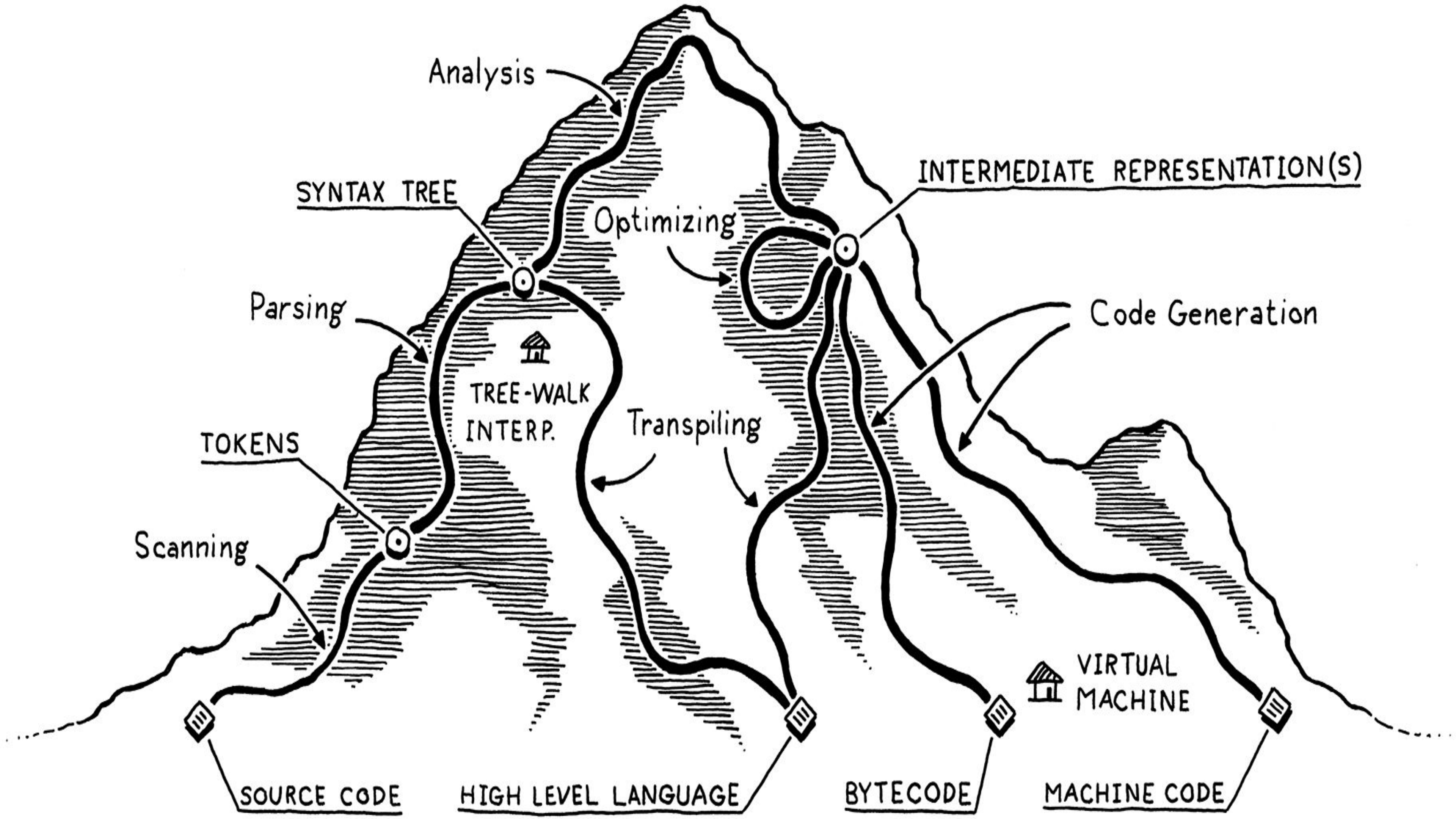
Progress



Introduction Overview Lexing Parsing Evaluation

What we intend to do today

1. Roadmap of the translation process


Analysis
SYNTAX TREE
Optimizing
INTERMEDIATE REPRESENTATION(S)
Parsing
Code Generation
TREE-WALK
INTERP.
Transpiling
TOKENS
Scanning
VIRTUAL
MACHINE
SOURCE CODE
HIGH LEVEL LANGUAGE
BYTECODE
MACHINE CODE

What we intend to do today

1. Roadmap of the translation process
2. Our language specification!

What we intend to do today

Our language specification

- Assignment statements

```
x := 1
```

- Conditional statements:

```
if x = 1 then
  y := 2
else
  y := 3
end
```

What we intend to do today

Our language specification

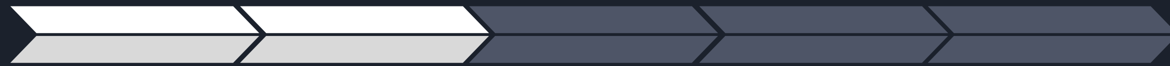
- While statements:

```
while x < 10 do
  x := x + 1
end
```

- Compound statements (separated by semicolons):

```
x := 1;
y := 2
```

Progress



Introduction Overview Lexing Parsing Evaluation

Lexing

1. Crash course on regular expressions

Lexing

Regular Expressions

Wikipedia says

A regular expression is a sequence of characters that define a search pattern

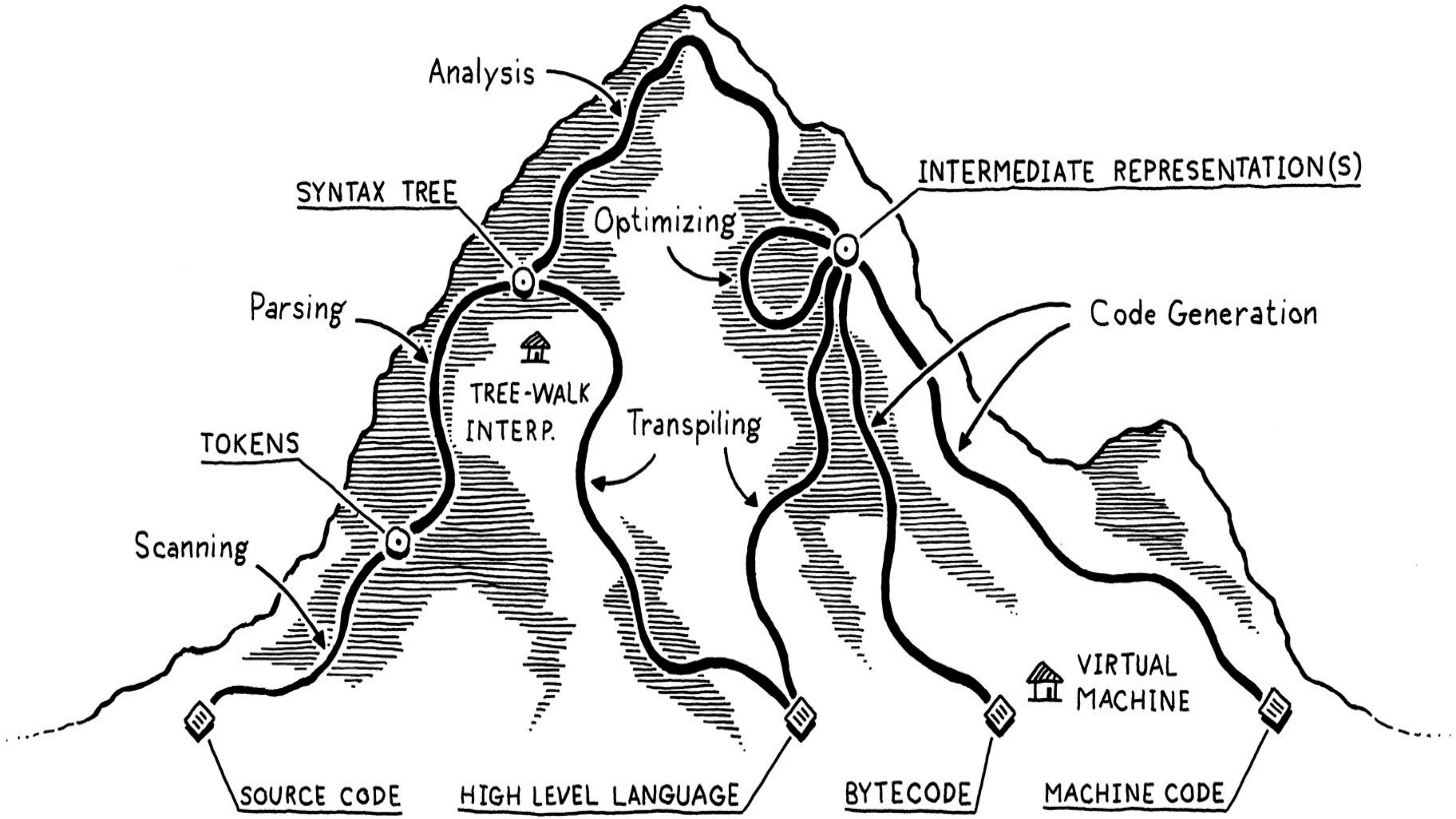
```
'[0-9]+'                        INT
Matches: 666
'[A-Za-z][A-Za-z0-9_]*'         ID
Matches: This_Talk_Sucks_123
'#[^\n]*'                       COMMENTS
Matches: # This is a splendid comment
```

v	a	r		a	v	e	r	a	g	e	=	(	m	i	n	+	m	a	x	)	/	2	;

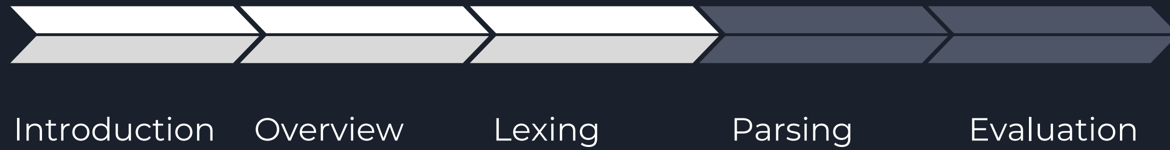
var	average	=	(	min	+	max	)	/	2	;

Lexing

1. Crash course on regular expressions
2. Demo


Analysis
SYNTAX TREE
Optimizing
INTERMEDIATE REPRESENTATION(S)
Parsing
Code Generation
TREE-WALK INTERP.
Transpiling
TOKENS
Scanning
VIRTUAL MACHINE
SOURCE CODE
HIGH LEVEL LANGUAGE
BYTECODE
MACHINE CODE

Progress


Introduction
Overview
Lexing
Parsing
Evaluation

Parsing

1. Grammars and Abstract Syntax Trees

Parsing

Grammars

Wikipedia says

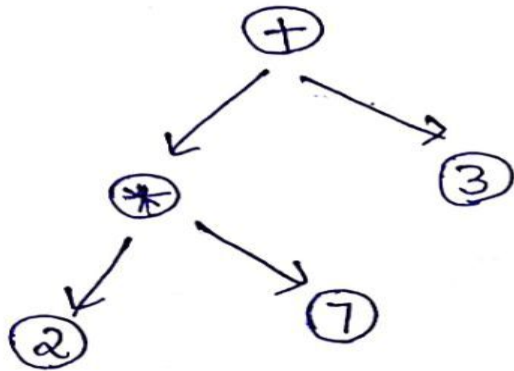
> *A grammar is a set of production rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax.*

```
Expr   -> Expr + Term
        | Expr - Term
        | Term

Term   -> Term * Factor
        | Term / Factor
        | Factor

Factor -> ID | NUMBER
```

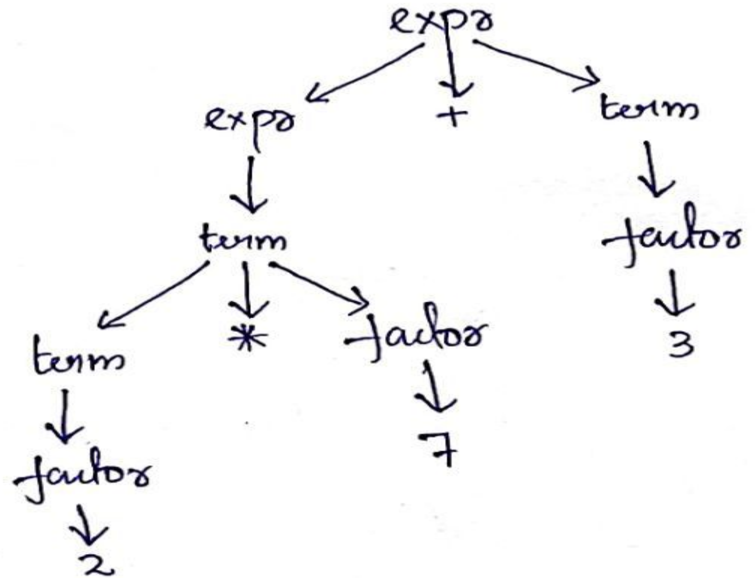
Matches: `x + 10 or 21 or x or 12 +66`

2 * 7 + 3

AST



Parse Tree



Parsing

Grammar for our language

Statements can contain both arithmetic and Boolean expressions. There are four kinds of statements:

- Assignment
- Compound
- Conditional
- Loop

```
CompoundStatement -> Statement ; CompoundStatement
                   | Statement
```

Parsing

Grammars

```
Statement       -> Assignment | IfStatement | WhileStatement
Assignment      -> VAR := AExp
                 | VAR := BExp

IfStatement     -> if BExp then
                     CompoundStatement
                   else
                     CompoundStatement
                  end

WhileStatement  -> while BExp do
                     CompoundStatement
                   end
```

Parsing

Grammars

An arithmetic expression can take one of three forms:

- Literal integer constants, such as 42
- Variables, such as x
- Binary operations, such as x + 42. These are made out of other arithmetic expressions.

```
AExp -> AExp Op AExp
      | VAR
      | INT

Op    -> + | - | * | /
```

```
Matches: x + 10 or 21 or x or 12 + 66
```

Parsing

Grammars

There are four kinds of Boolean expressions.

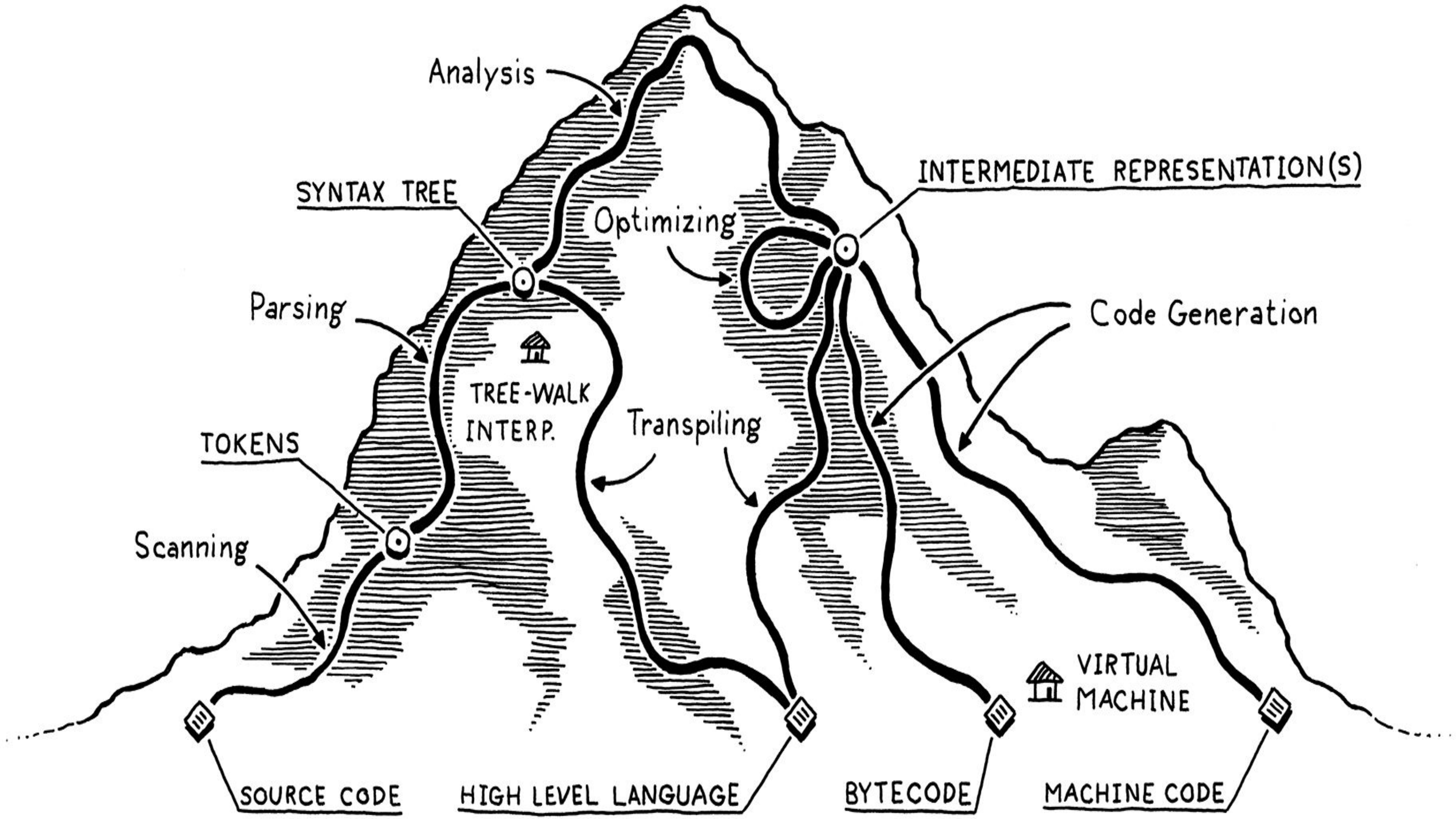
- Relational expressions (such as x < 10)
- **AND** expressions (such as x < 10 and y > 20)
- **OR** expressions
- **NOT** expressions

```
BExp -> BExp Op BExp
      | NOT Bexp
      | True | False

Op   -> AND | OR | >= | <= | < | >
```

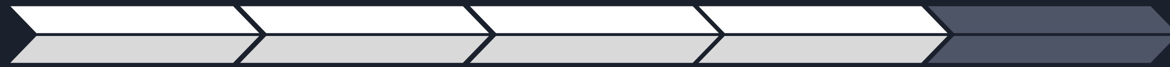
```
Matches: A OR B, A AND B, NOT A
```

Parsing

1. Grammars and Abstract Syntax Trees
2. Demo



Analysis
SYNTAX TREE
Optimizing
INTERMEDIATE REPRESENTATION(S)
Parsing
Code Generation
TREE-WALK
INTERP.
Transpiling
TOKENS
Scanning
VIRTUAL
MACHINE
SOURCE CODE
HIGH LEVEL LANGUAGE
BYTECODE
MACHINE CODE



Introduction Overview Lexing Parsing Evaluation

Evaluating expressions

01

```
class AssignStatement(Statement):
    ...
    def eval(self, env):
        value = self.aexp.eval(env)
        env[self.name] = value
```

02

```
class CompoundStatement(Statement):
    ...
    def eval(self, env):
        self.first.eval(env)
        self.second.eval(env)
        ...
```

03

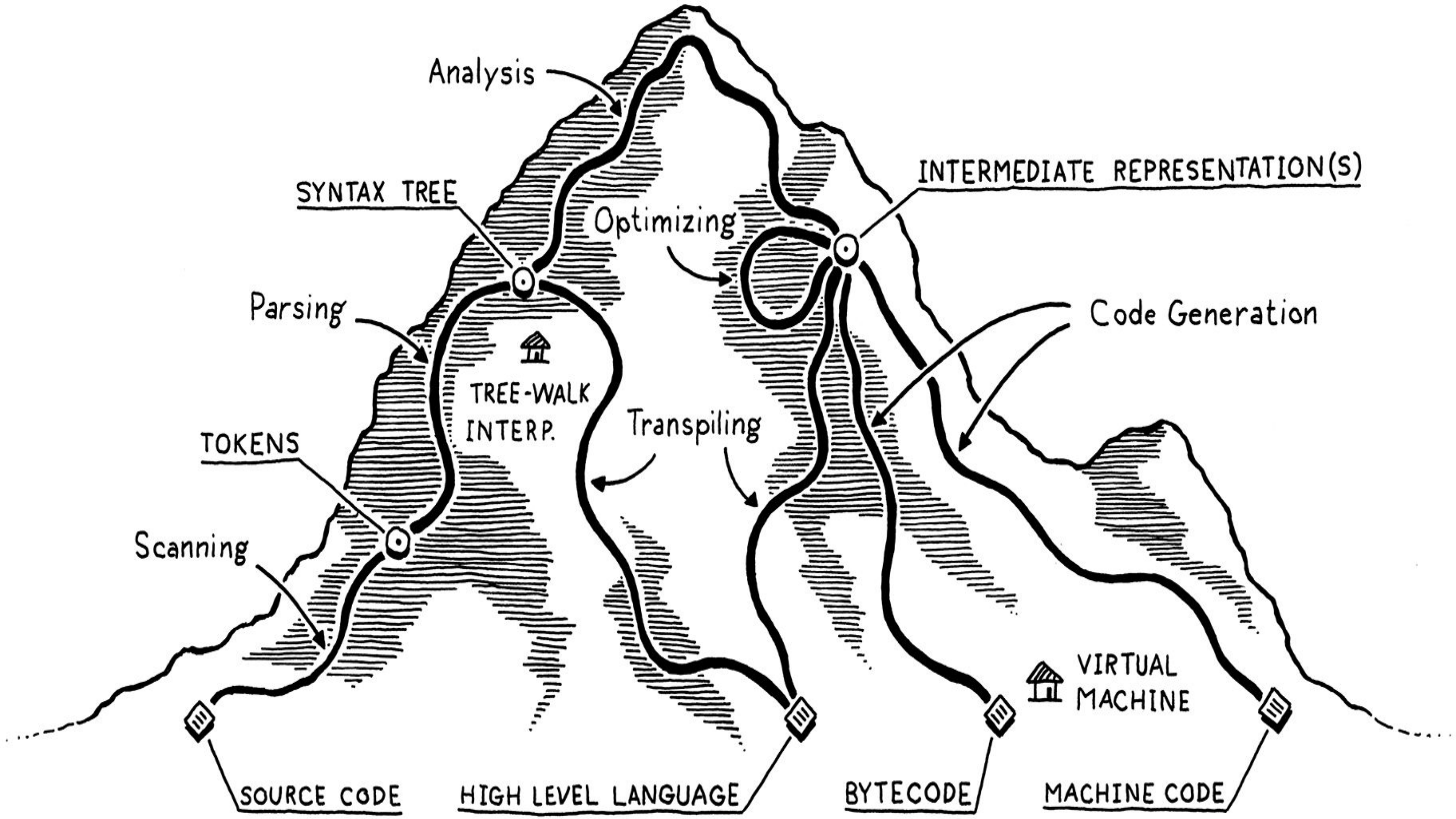
```
class IfStatement(Statement):
    ...
    def eval(self, env):
        condition_value = self.condition.eval(env)
        if condition_value:
            self.true_stmt.eval(env)
        else:
            if self.false_stmt:
                self.false_stmt.eval(env)
```

04

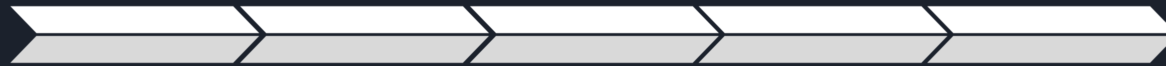
```
class WhileStatement(Statement):
    ...
    def eval(self, env):
        condition_value = self.condition.eval(env)
        while condition_value:
            self.body.eval(env)
            condition_value =
self.condition.eval(env)
```

Let’s calculate factorials!

```
n := 5;
p := 1;
while n > 0 do
  p := p * n;
  n := n - 1
end
```


Analysis
SYNTAX TREE
Optimizing
INTERMEDIATE REPRESENTATION(S)
Parsing
Code Generation
TREE-WALK
INTERP.
Transpiling
TOKENS
Scanning
VIRTUAL
MACHINE
SOURCE CODE
HIGH LEVEL LANGUAGE
BYTECODE
MACHINE CODE

Progress: We’re done!


Introduction
Overview
Lexing
Parsing
Evaluation

Thank you!

Credits

- http://www.jayconrod.com/posts/40/a-simple-interpreter-from-scratch-in-python-part-1
- craftinginterpreters.com